

Predicting the Performance of Scientific Applications on Distributed Memory Multiprocessors

D. Abramson

School of Computing and Information Technology
Griffith University
Kessels Rd, Brisbane,
Queensland, 4111
davida@cit.gu.edu.au

Abstract

Porting scientific applications to parallel machines is one of the major challenges for scientists and computing professionals over the coming years. Existing codes span many thousands of lines and are often converted for parallel execution by someone other than the original author. Good tools are required to assist this process as much as possible. Two key design parameters that must be addressed on distributed memory computers are the way that parallel data structures are distributed across the machine and the way that processes are allocated to processors. This paper describes a tool called DCompose, which allows a user to rapidly determine the effect of different data decomposition and process allocation strategies. Using DCompose, important design decisions can be made before the translation of the program is started, and before the target platform is chosen.

1. Introduction

This paper addresses two critical issues in porting sequential scientific applications from uni processors to distributed memory multiprocessors, namely, algorithmic decomposition and data decomposition. Most existing scientific codes have been developed in a piecemeal manner on uni processors, and are then converted for parallel execution. Depending on the target platform, this may involve the addition of a small number of parallelisation directives or a major rewrite in another language. Regardless of the amount of effort involved, it is desirable to make some major design decisions before the code is modified. In many cases, the parallel hardware is an expensive shared resource, thus it is desirable that some of the design be performed on the original uni processor version of the program.

In this paper we describe the processes of algorithmic decomposition and data decomposition as they relate to many existing scientific codes, and explore the effect they

have on performance. We then describe a tool (called DCompose) which allows the programmer to experiment with different design criteria. This tool allows the user to instrument an existing sequential application augmented with some directives. The output of the instrumentation process is a visualisation of the workload distribution and the data communication activity. This is produced without the need to add any explicit communications commands. The user then alters the directives to minimise communication whilst achieving a good workload distribution across the processors.

The program is then ported to the parallel platform using the parallel languages and performance monitoring tools provided by the computer vendor. DCompose differs from almost all commercially available profilers in that it provides early design assistance before the program is ported to a particular machine. It also provides source level information rather than global program statistics, so that it is possible to trace performance problems to the code which generates them.

The paper begins with a discussion on techniques for decomposing scientific algorithms, followed by a description of the possible data decompositions. It then describes DCompose, as a tool for measuring the efficiency of various decompositions. Finally, DCompose is illustrated by its use in assisting the porting of a photo chemical pollution model to a distributed memory multiprocessor.

2. Algorithmic Decomposition

In many scientific programs the major source of concurrency is through the unravelling of loops. Using this scheme, loops are partitioned into sections, each of which is executed by one processor. Figure 1 shows a sample loop, in which the index is varied from 1 to 100. On a sequential machine, the loop index is initialised to 1 and incremented until it reaches 100. On a four processor parallel machine, the loop is *sliced* into four loops, each

with 25 iterations. Figure 1 shows each processor executing the loop with different upper and lower bounds.

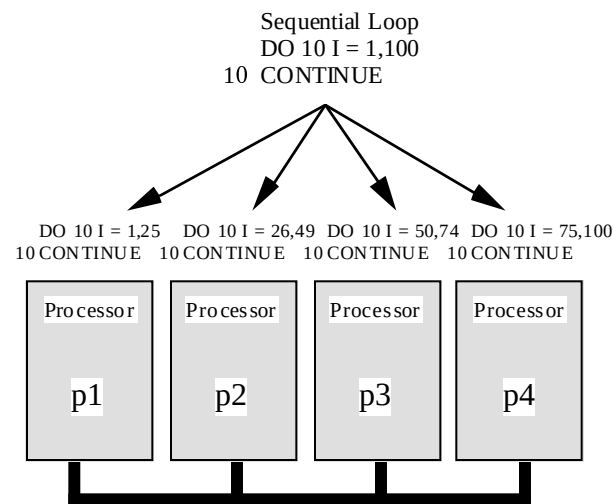


Figure 1 - Slicing of loop across processors

There are two main ways of implementing loop slicing, namely *static* and *dynamic*. In static slicing the compiler allocates the loop bounds for each of the worker processors when the program is compiled. In dynamic slicing, each processor takes work from a central work queue. In this way the loop indexes are allocated to free processors until the loop terminates. Dynamic allocation has the advantage that the workload can be balanced when different loop iterations take varying amounts of time to execute.

3. Data Decomposition

In order to map an algorithm onto a distributed memory multiprocessor, it is necessary to partition the key data structures so that they are distributed across the multiple memory modules. Through the communication network they are accessible to more than one processor. This is relatively easy for applications which do not require much co-ordination between tasks, however, choosing an optimal layout for distributed data structures in a large scientific code can be difficult, especially when the access patterns change as the program executes.

A number of new dialects of FORTRAN have been devised to specify different decompositions for data structures. High Performance FORTRAN and FORTRAN D [1] both include additional statements to allow structures to be distributed across one or more of their indexes. A user can specify a block decomposition using the keyword BLOCK, a cyclic decomposition using the keyword CYCLIC, no decomposition using * and a combination of

block and cyclic with a BLOCK_CYCLIC keyword. Figure 2 shows a sample BLOCK and CYCLIC decompositions for a one dimension data structures. In this example, array A is distributed across four processors called p1 through p4. It can be seen that BLOCK causes the structure to be broken into a number of contiguous regions which are then allocated to processors. CYCLIC causes elements of the array to be allocated in a cyclic manner. BLOCK_CYCLIC combines the two. For more details the reader is referred to the FORTRAN D users manual.

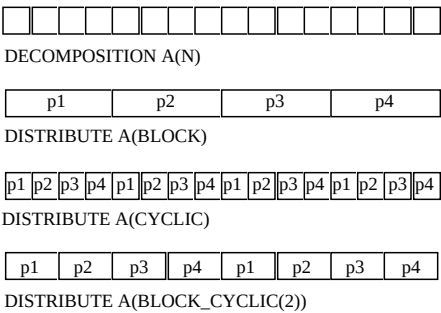


Figure 2 - Data decomposition of array A

4. A Tool for Predicting Performance

DCompose is a tool which was developed to measure the expected performance of various different work load allocation strategies and of different data decompositions. DCompose instruments an executing program and records the amount of inter processor traffic caused by the data decompositions and the amount of processor time attributed to pseudo processors. The strategy adopted for instrumenting the sequential code is shown below in Figure 3. The source program is augmented by directives which indicate which data structures are to be shared between processes, and what data decomposition should be used. It is also modified to indicate how work is allocated to processes; typically this is through loop unravelling, and thus the loop iteration value must be mapped onto a pseudo processor number.

Source code is compiled using the standard FORTRAN or C compiler, but instead of generating object code, a switch is used to produce assembler. The assembler is then automatically processed to allow all memory references to be traced. When the program is executed, a specially linked run time system keeps track of the required statistics and a summary trace file is generated. A visualisation tool then presents the results to the programmer, who may further modify the code and rerun the instrumented code.

4.1 Underlying model of execution

DCompose is used to assist a user in decomposing data for an arbitrary distributed memory machine. The final program may execute on a message passing platform, in which case message passing calls must be inserted, or on a distributed shared memory machine, in which case the decompositions must simply be specified. The distributed shared memory model is used to measure the traffic generated by different decompositions, because it requires the minimum alteration to the sequential source.

Using this model the user must

- declare which data structures are to be shared between processors;
- give data decompositions for each structure;
- denote the form of workload allocation.

Since all of this information is declarative, it does not affect the execution of the code. It is only used by the run time system to determine the effect of the various allocations.

4.2 Recording inter processor memory traffic

In order to compute the amount of inter processor traffic that results from a program execution, the code is augmented by the programmer (or another automatic parallelisation tool like FORGE-90 [2]) with directives to indicate which structures must be globally accessible, and how they are to be partitioned. Because the program is run on a uni-processor, it is only necessary to compute the target memory module number from the address, and maintain statistics on the number of accesses between pseudo processors and memories. The memory reference is still executed on the uni-processor memory and the program behaves in exactly the same way as if it was un-instrumented.

The assembler code is modified by a separate filter program so that before every *load* and *store* instruction the effective address is passed to a routine for interpretation. If the address is in the globally shared region then it is further examined to determine which data structure it belongs to, and the array indexes are computed. These are then used to calculate the target memory module number based on the partitioning which has been specified for that structure. All of the information about each data structure, such as its start address, size, number of indexes, maximum size of

each index and data partitioning directive for each index, is stored in a special table, which is initialised when the program is started. In this way the run time system can look up the values as required.

The source pseudo processor number is set by a number of directives in the code. In the case that loop unravelling is used to expose concurrency, the processor identity can be computed from the loop control variable. For every parallel loop a routine is called to set the processor identity, which is then used by the run time system. This procedure allows the program to appear to execute on more than one processor whilst still running on a uni-processor.

4.3 Measuring workload allocation

It is possible to measure the effectiveness of the workload allocation by keeping track of the amount of time spent by each processor. The scheme described in the previous section allows the run time system to detect when a process switch occurs. By keeping track of the amount of processor time between each processor switch, it is possible to build up a profile of execution time for each processor, providing a measure of the workload allocation.

Since the run time system slows down the execution of shared memory references, it is desirable to gather then processor time statistics on an uninstrumented program. This operation is extremely fast and thus it is not a serious disadvantage to have to execute the program twice.

4.4 The Directives

In order to keep DCompose language independent, the decomposition and process allocation strategies are specified using subroutine calls. The following subroutines are provided:

- `INITTRACING(start,end)`

This routine declares the region of shared memory. The user places all shared variables in a contiguous memory segment. In FORTRAN this is achieved by placing all shared variable in common. *Start* is the first variable, and *end* is the last one.

- `DISTRIBUTE(start, end, x_size, x_dist, y_size, y_dist, z_size, z_dist,type,'name')`

This routine declares the data decomposition to use for each data structure. The *start* and the *end* define the address bounds of the variable. *X_size*, *y_size*

and *z_size* show the actual sizes of the data declaration. If a structure is two dimensional then the *z_size* is set to zero, and if the structure is one dimensional then *y_size* is also set to zero. *X_dist*, *y_dist* and *z_dist* give the possible decompositions for each of the indexes. Legal values are 'BLOCK', 'CYCLIC' and '*'. *Type* describes the base type of the variable, and *name* contains a string descriptor for the variable. This name is usually set to the actual name of the variable. DISTRIBUTE can be called more than once per data structure, making it possible to redistribute the data dynamically.

- SETPROCESSOR(*processor*)

This routine sets the pseudo processor number in the run time system. If loop unravelling is being used to allocate the work to processors, then an expression based on the loop index is used to compute the identity of the processor which would have generated that section of the loop. This routine must be called each time the loop index is altered.

- UNSETPROCESSOR()

This routine resets the pseudo processor to a null value. This makes it possible to detect accesses to shared variables for regions of code which have not been parallelised, allowing for incremental parallelisation of the code.

- DUMPTRACE()

This routine dumps the trace data to a file for later visualisation. It is usually only called once at the end of a run, but it is possible to call it multiple times to capture different slices of the execution.

4.5 What does DCompose display?

One of the features of DCompose is the way in which it presents the results to the programmer. Figure 4 shows a sample screen from DCompose. The user is presented with a grey scale map (or false colour if available) of the inter processor traffic that results from the given data decomposition and code partitioning. A two dimensional matrix indexed by processor number and memory module number shows the number of references between any pair of processors and memory. By clicking with the mouse on a particular matrix cell, the user is able to observe the source of that traffic. A new window is opened showing all of the sources of the traffic, and by clicking on one particular source, the original code is displayed. This is shown in Figure 5.

It is possible to filter the information by selecting only certain routines, data structures or lines of code. This filtering makes it possible to determine the source of particular inter processor communication. The filters are particularly useful when a data structure is addressed from more than one point in the code with different access patterns. By filtering out the different sources it is possible to determine which accesses cause non-local memory references. The user can also select only reads or only writes, thus it is possible to see where data structures are initialised.

An optional cache module in the run time system allows the user to cache locations as they are accessed. This makes it possible to determine the effectiveness of a real cache memory in the target processor, because the memory traffic can be compared to that generated without caching. The effect of caching is particularly evident when global structures are initialised in the mainline, and are then accessed (often through COMMON) in subroutines. In a distributed memory machine these structures can be initialised once and then multiple copies can be distributed, thus eliminating all future interprocessor memory traffic when they are accessed.

The workload is displayed in a separate window. The amount of time accrued by each processor is shown using a grey scale. Using this feature it is possible to observe any major differences in the amount of work performed by each processor. At present this feature of DCompose has not been fully implemented, and thus the displays in Figures 4 and 5 do not show any information in their workload windows.

More details of DCompose can be found in [3]

4.6 What DCompose does not display

There are a number of ways of producing the information outlined at the beginning of this section. A full discrete event simulation of a multiprocessor system has the ability to provide detailed accounting and timing information. However, full simulations are often very slow, and thus hinder rapid prototyping of different decomposition strategies. Also, they provide far more information than is required to access the effectiveness of the decompositions.

The scheme which has been adopted by DCompose is to augment a sequential execution with sufficient trace information to count the relative frequencies of the various memory accesses. Consequently, the system cannot

provide any information about the timing between memory references, and thus effects like memory hot-spots cannot be observed. Also, differences in workload allocation

which depend on synchronisation between processors cannot be measured.

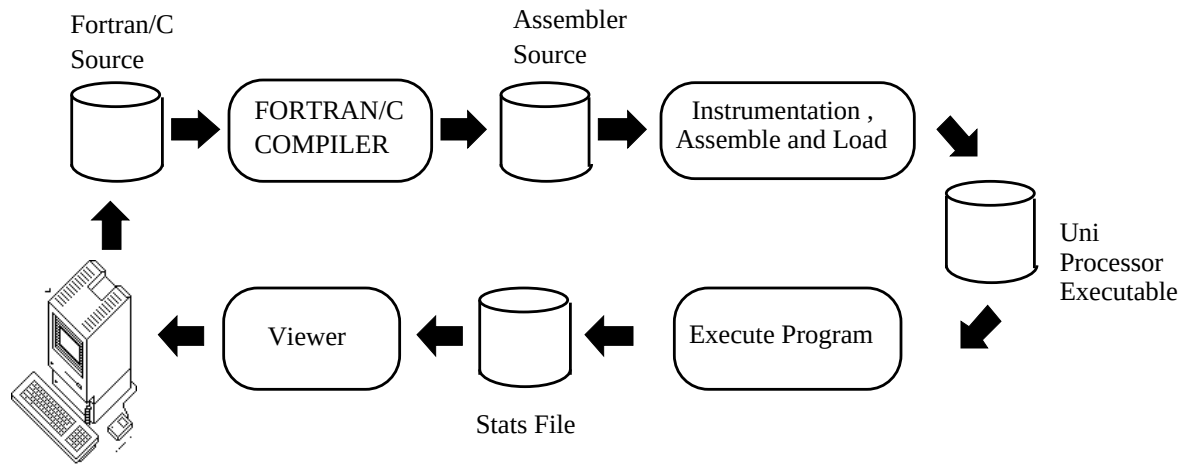


Figure 3 - instrumentation process

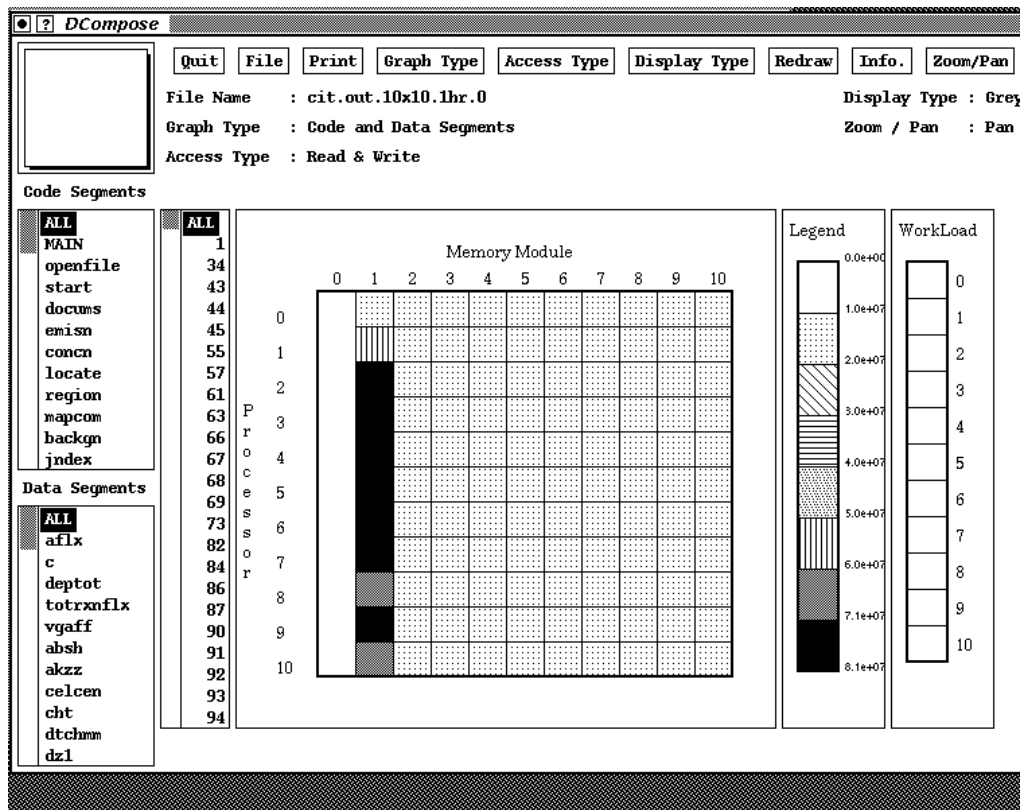


Figure 4 - Sample Main screen of DCompose

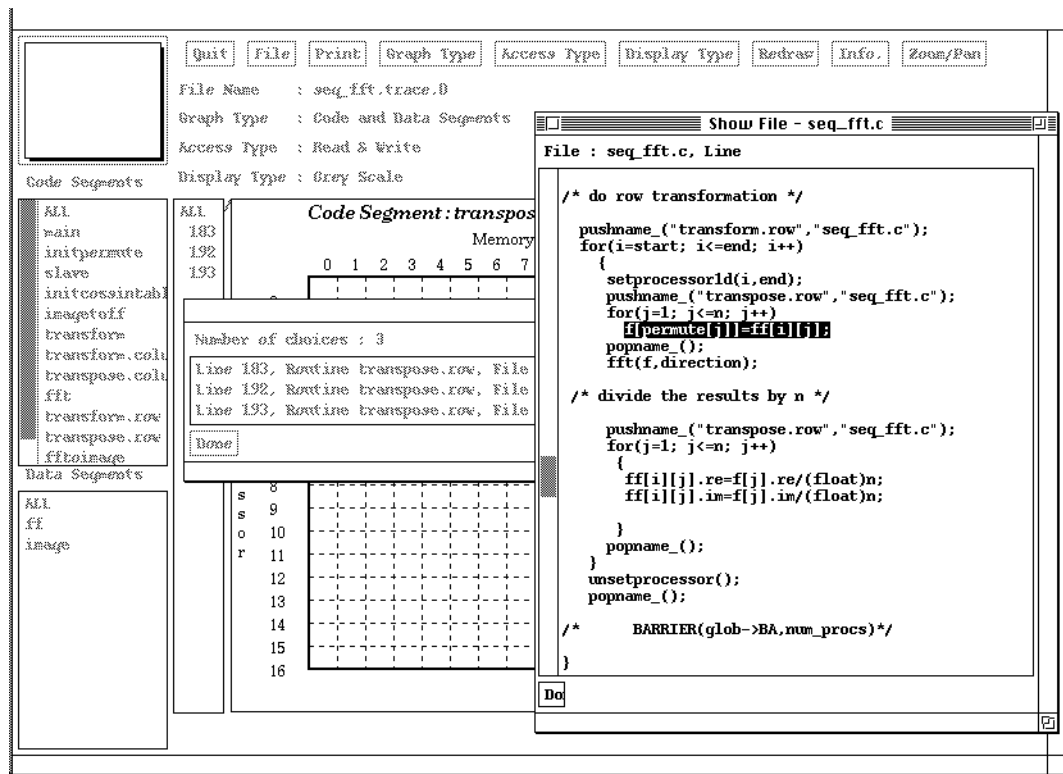


Figure 5 - Showing source code using DCompose

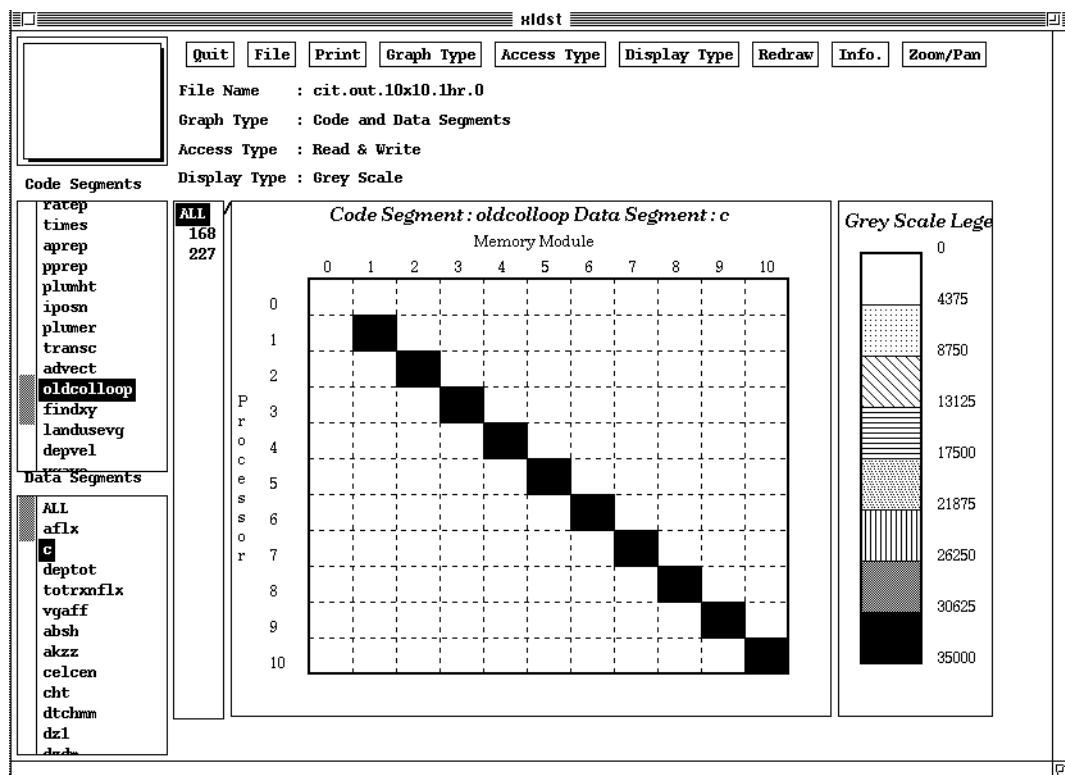


Figure 6 - Distribution across x index

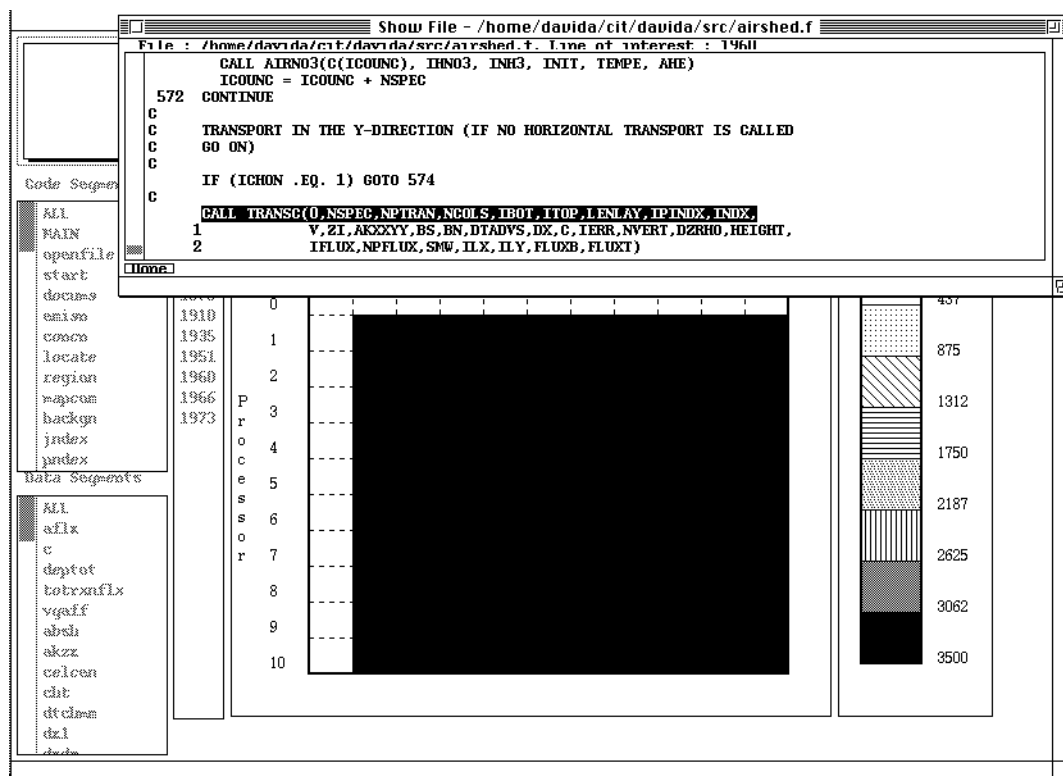


Figure 7 - Effect of x decomposition during x transport.

5. Parallelising the CIT Smog Model

In this section we demonstrate the use of DCompose in the parallelisation of a photo chemical pollution model, developed by McRae at CalTech [4][5]. This model uses precomputed wind vectors to disperse pollutants over a three dimensional grid. The program solves the ordinary differential equations that describe the chemistry which occurs when the pollutants are exposed to ultra violet radiation, and it produces concentrations for all species over the grid. This is depicted in Figure 8.

From a parallelisation viewpoint the program is interesting because it manipulates some very large data structures which must be partitioned across the memories, as well as some smaller global constants which must be replicated in each memory. Also, the program is composed of two main phases which are executed in cycles; one in which the chemical rate equations are solved, and another in which the chemical species are dispersed between grid cells. Thus, one data layout is not suitable for the entire program and the layout needs to be altered each time the program executes one main cycle.

When the program is computing the new concentrations of the species, many of the data structures can be partitioned in such a way that each processor handles some subset of the grid cells. In this way, no inter processor communication is necessary. However, when the species are dispersed, it is necessary to move data between grid cells. In order to perform this operation in a parallel manner, the grid must be partitioned into rows when the dispersion is in the x direction, and into columns when the dispersion is in the y direction.

One of the most important data structures is the concentration vector C. C is logically a four dimensional array indexed by the x, y and z co-ordinates of the grid cell and the species number. In practice, C is implemented as a single dimensional array with an expression for the index based on the grid cell number and the species number. Logically the following FORTRAN declaration applies to C:

```
REAL C(X,Y,Z,SPECNO)
```

There are a number of ways of distributing C. Each computation in the chemistry phase works on the grid cells independently, and thus a decomposition based on

the cell number is effective at removing any interprocessor traffic. Thus, the following statement is effective, as is shown in Figure 6.

DISTRIBUTE C(BLOCK,*,*,*)

This decomposition is also effective during y transport because the species are moved in the y direction but the x values are independent. However, Figure 7 shows the effect of the computation on the x transport. As a consequence the following declaration is required:

DISTRIBUTE C(*,BLOCK,*,*)

An alternative is to distribute the vector along the species index because all species can be transported concurrently. Thus the following declaration is effective:

DISTRIBUTE C(*,*,*,BLOCK)

The conclusion of this discussion is that no decomposition is effective for all phases of the computation. Interprocessor communication is required to re-distribute the data between chemistry and transport. This is clearly illustrated by the DCompose output in Figures 6 and 7.

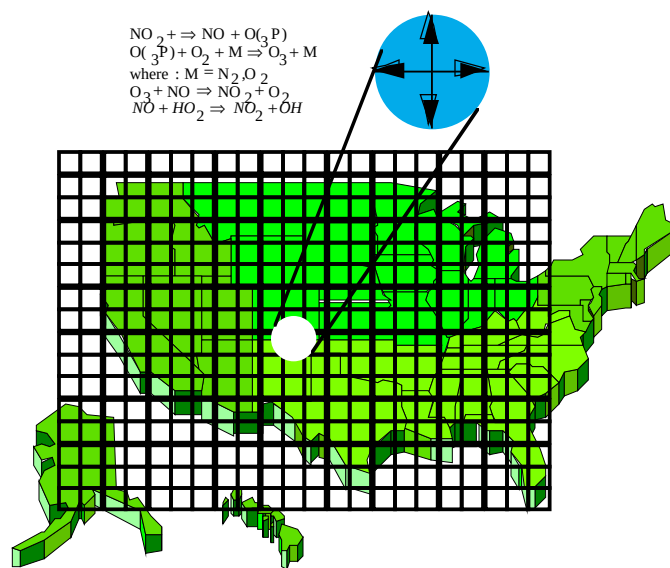


Figure 8 - computational grid structure in CIT model

Acknowledgements

Thanks go to Gavin Cameron who implemented much of the DCompose code. This work was sponsored by the Division of Information Technology of the Commonwealth Scientific and Industrial Research Organisation.

References

- [1] Fox, G. et al, (1992) "The FORTRAN D Language Specification", Internal Report, NPAC, Syracuse University, NY 13244., January 1992
- [2] John M. Levesque, "FORGE 90 and High Performance Fortran (HPF)", Presented at the NATO Advanced Research Workshop on Software for Parallel Computation, June 1992.
- [3] Abramson, D., Cameron, G. "DCompose: A Tool for Measuring Data Decomposition on Distributed Memory Multiprocessors", to appear in Current and Future Trends in Parallel and Distributed Computing, Ed Albert Zomaya, Chapman & Hall Publishing Company.
- [4] McRae, G.J., Goodwin, W. R. and Seinfeld, J.H. (1982) "Development of a Second Generation Mathematical Model for Urban Air Pollution: I Model Formulation", Atmospheric Environment, 16, pp 679-696., 1982
- [5] Goodwin, W.R., McRae, G.J. and Seinfeld, J.H. (1979) "An Objective Analysis Technique for Constructing Three-Dimensional Urban-Scale Wind Fields", Journal of Applied Meteorology, 19, 98-108, 1979